The SugarCubes Tool Box

Frédéric Boussinot, Jean-Ferdy Susini

INRIA
INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

The SugarCubes Tool Box

Frédéric Boussinot, Jean-Ferdy Susini

N° 3247

Septembre 1997

THÈME 1

ISSN 0249-6399

INRIA
SOPHIA ANTIPOLIS

The SugarCubes Tool Box

Frédéric Boussinot, Jean-Ferdy Susini

Thème 1 — Réseaux et systèmes
Projet MEIJE

Rapport de recherche n°3247 — Septembre 1997 — 29 pages

Abstract:

SugarCubes are a set of Java classes for implementing software systems such as:

- Event based systems, especially those where events are instantly broadcast throughout the system. Communicating in this framework is like in radio transmissions, where emitters send information that is immediately received by all receivers.
- Concurrent systems, in particular threadless ones. Here, parallelism is a logical programming construct to implement activities which are supposed to proceed concurrently, and not in sequence.
- Reactive systems, which continuously react to activations from an external source.

SugarCubes are used to implement reactive scripts and icobj programming on top of the Java language. They are available at http://www.inria.fr/meije/rc/SugarCubes.

Key-words: Reactive programming, Java, Broadcast Events

(Résumé : tsvp)

EMP-CMA, B.P. 207, F-06904 Sophia Antipolis cedex

Unité de recherche INRIA Sophia Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA ANTIPOLIS Cedex (France)
Téléphone : 04 93 65 77 77 – Télécopie : 04 93 65 77 65

Les SugarCubes

Résumé :

Les SugarCubes sont un ensemble de classes Java pour implémenter les systèmes comme :

• Les système événementiels, en particulier ceux dans lesquels les événements sont diffusés. La communication est alors analogue à la radio dans laquelle les émetteurs envoient des informations qui sont instantanément reçues par tous les récepteurs.

• Les systèmes parallèles, en particulier ceux qui n'utilisent pas de threads. Dans ce cadre, le parallélisme est uniquement un moyen logique d'implémenter des activités qui doivent s'exécuter en concurrence et non en séquence.

• Les systèmes réactifs qui réagissent de manière continue aux activations de leur environnement.

Les classes SugarCubes sont utilisées pour implémenter les scripts réactifs et la programmation par icobjs au dessus de Java. Elles sont disponibles en http://www.inria.fr/meije/-rc/SugarCubes.

Mots-clé : Parallélisme, Java, Diffusion d'événements

The SugarCubes Tool Box[1]

Contents

[1]Work supported by France Télécom/Cnet and Soft Mountain

1 Introduction

SugarCubes are a set of Java classes for implementing software systems such as:

- *event based systems*, especially those where events are instantly broadcast throughout system. Communicating in this framework is like in radio transmissions, where emitters send information that is instantaneously received by all receivers. This communication paradigm gives a very modular way of system structuring. For example, adding new receivers to a system is totally transparent for the existing components (which is not the case with other communication mechanisms like message passing or "rendez-vous").

- *parallel systems*, in particular threadless ones. Here, parallelism is a logical programming construct to implement activities which are supposed to proceed concurrently and not in sequence. Such parallel activities need not to be executed by distinct threads, but instead may be automatically "interleaved" to get the desired result. This avoids well-known problems related to threads (thread-unsafe classes).

- *reactive systems*, which continuously react to activations. A natural way of programming these systems is by combining reactive instructions whose semantics are defined by reference to activation/reaction couples, also named *instants*. The end of the reaction provoked by an activation gives a natural way for determining stable states, where an instruction or a system is only waiting for the next activation to resume execution. The existence of stable states is of major importance for agent migration over the network.

SugarCubes basically introduce the notion of a global logical clock, and system runs are decomposed into *instants* shared by all parallel components. This is the reactive paradigm whose description can be found at `http://www.inria.fr/meije/rc`. Several related papers are available at this URL.

Presently, two main applications are implemented using SugarCubes:

- rsi-java which is the implementation of reactive scripts on top of Java. Reactive scripts will give a very flexible and powerful means to program over Internet.

- Icobjs (for "*icon objects*") define a new, fully graphical way of programming. A demo based on icobjs is available on the Web at `http://www.inria.fr/meije/-rc/WebIcobj`. In this demo a little robot is dynamically programmed to bypass an obstacle.

SugarCubes[2] are freely distributed as a tool box for reactive programming in Java(the presently distributed version is the beta version v1.0).

In this paper we present SugarCubes and give most part of the code for each of them.

[2] Why the name SugarCubes? because many people like to add some sugar in their Java...

2 The Main SugarCubes Classes

The two main SugarCubes classes are `Instruction` and `Machine`. `Instruction` defines reactive instructions which are defined with reference to instants, and `Machine` defines reactive machines which run reactive instructions.

2.1 Instructions

An instruction can be activated (`activ` method), reset (`reset` method), or forced to terminate (`terminate` method). Each activation returns `TERM`, `STOP`, or `SUSP`, three return codes defined in the `ReturnCodes` interface:

- `TERM` (for "terminated") means that execution of the instruction is completed. Thus, activating it at following instants will have no effect and will again return `TERM`.
- `STOP` (for "stopped") means that execution of the instruction is stopped at a stable state for the current instant, and could progress at the next instant upon activation.
- `SUSP` (for "suspended") means that execution of the instruction has not reached a stable state for the current instant, and has to resume during that instant. This is for example the case when awaiting for an event not yet generated (see section 5): execution suspends to give other components the opportunity to generate it.

A call to the `terminate` method forces an instruction to completely terminate and therefore to return `TERM` when activated. A call to the `reset` method resets the instruction in its initial state.

`Instruction` implements `Cloneable` and thus instruction clones are available by the `clone` method.

Finally, equality of two instructions (`equals` method) and the `toString` method to print an instruction, are implemented.

Instruction Class

The `Instruction` class has the following structure :

```
abstract public class Instruction
implements ReturnCodes, Cloneable
{
  protected boolean terminated = false;

  public void reset(){ terminated = false; }
  final public void terminate(){ terminated = true; }
  final public boolean isTerminated(){ return terminated; }

  abstract protected byte activation(Machine machine);
  final public byte activ(Machine machine)
```

```
  {
    if (terminated){ return TERM; }
    byte res = activation(machine);
    if (res == TERM){ terminated = true; }
    return res;
  }

  public boolean equals(Instruction inst){
    return this.getClass() == inst.getClass();
  }
  abstract public String toString();
  public Object clone(){ ... }
}
```

Note that `activ` and `activation` have a parameter which is the reactive machine running the instruction. Reactive machines are described in section 2.2.

UnaryInstruction Class

`UnaryInstruction` is an abstract class which extends `Instruction` and has a body which is also an `Instruction`:

```
abstract public class UnaryInstruction extends Instruction
{
  protected Instruction body;

  public void reset(){ super.reset(); body.reset(); }

  public boolean equals(Instruction inst){
    return  super.equals(inst) &&
            body.equals(((UnaryInstruction)inst).body);
  }

  public Object clone()
  {
    UnaryInstruction inst = (UnaryInstruction)super.clone();
    inst.body = (Instruction)body.clone();
    return inst;
  }
}
```

BinaryInstruction Class

`BinaryInstruction` is an abstract class which extends `Instruction` and has two components `left` and `right` which are also instructions.

```
abstract public class BinaryInstruction extends Instruction
{
  protected Instruction left, right;

  public void reset(){
    super.reset(); left.reset(); right.reset();
  }
  public boolean equals(Instruction inst){
    return  super.equals(inst) &&
            left.equals (((BinaryInstruction)inst).left) &&
            right.equals(((BinaryInstruction)inst).right);
  }
  public Object clone()
  {
    BinaryInstruction bin = (BinaryInstruction)super.clone();
    bin.left  = (Instruction)left.clone();
    bin.right = (Instruction)right.clone();
    return bin;
  }
}
```

2.2 Machines

A reactive machine of the `Machine` class runs a program which is an instruction. Initially programs are the `Nothing` instruction (defined in section 3.1) which does nothing and terminates instantaneously.

Basically, `Machine` detects the end of the current instant, that is when all parallel instructions of the program are terminated or stopped. There is another case where the end of the current instant is detected: when there is no new move in the program after two activations of it; in this case, there is no hope that new activations will change anything to the situation and the end of the instant can be safely set. In this case the machine activates the program once more, after having set the end of instant flag, to let suspended instructions stop. Code for this is (see the `activation` method of `Machine` below):

```
while ((res = program.activ(this)) == SUSP){
   if (move) move = false; else endOfInstant = true;
}
```

Two methods are used to manage the two flags `move` and `endOfInstant`: `newMove` which sets the `move` flag to indicate that something new happens in the program (thus, end of instant has to be postponed), and `isEndOfInstant` which tests the `endOfInstant` flag.

The `add` method adds a new instruction to the program; this new instruction is run in parallel with the previous program, using the `Merge` parallel instruction defined in section 3.3:

```
public void add(Instruction inst){
```

```
  program = new Merge(program,inst);
  newMove();
}
```

Note that `newMove` is called to let the new instruction execute during the current instant.

`Machine` contains an environment named `eventEnv` to deal with events (events are described in section 5).

`Machine` extends `Instruction` and thus, to execute the program for one instant simply means to activate the machine. The class `Machine` has the following structure:

```
public class Machine extends Instruction
{
  public Instruction program = new Nothing();
  protected EventEnv eventEnv = new EventEnv();

  protected int instant = 1;
  protected boolean endOfInstant = false, move = false;

  public void newMove()           { move = true; }
  public int currentInstant()     { return instant; }
  public boolean isEndOfInstant() { return endOfInstant; }

  public Object clone(){ ... }

  public void add(Instruction inst){
     program = new Merge(program,inst);
     newMove();
  }

  /** Methods to use the event environment. */
  public Event getEvent(String name){ ... }
  public boolean isGenerated(String name){ ... }
  public void generate(String name){ ... }
  public void putEvent(String name,Event event){ ... }

  protected byte activation(Machine machine){
     ...
    endOfInstant = move = false;
    while ((res = program.activ(this)) == SUSP){
       if (move) move = false; else endOfInstant = true;
    }
    instant++;
    ...
  }
}
```

3 Basic Instructions

In this section we introduce the basic instruction to stop and suspend execution for the current instant, the sequence and parallel instructions, atoms to execute basic actions, loops, and the closing statement.

3.1 Nothing, Stop, and Suspend Instructions

`Nothing` does nothing: it is introduced only as the initial program value.

```
public class Nothing extends Instruction
{
  final public String toString(){ return "nothing"; }
  final protected byte activation(Machine machine){return TERM;}
}
```

`Stop` stops execution for the current instant by returning `STOP`. However, the instruction terminates, thus activation will return `TERM` at the next instant.

```
public class Stop extends Instruction
{
  final public String toString(){ return "stop"; }

  final protected byte activation(Machine machine)
  {
    terminate();
    return STOP;
  }
}
```

`Suspend` suspends execution for the current instant by returning `SUSP`.

```
public class Suspend extends Instruction
{
  final public String toString(){ return "suspend"; }

  final protected byte activation(Machine machine)
  {
     terminate();
     return SUSP;
  }
}
```

3.2 Sequencing

`Seq` extends `BinaryInstruction` and implements sequencing. First the `left` instruction is activated; if it terminates the control goes to the `right` instruction.

```
public class Seq extends BinaryInstruction
{
  public Seq(Instruction left, Instruction right)
  {
     super.left = left;
     super.right = right;
  }

  final public String toString(){ return left + "; " + right; }

  final protected byte activation(Machine machine)
  {
     if (left.isTerminated()) return right.activ(machine);
     byte res = left.activ(machine);
     if (res != TERM) return res;
     return right.activ(machine);
  }
}
```

3.3 Parallelism

The `Merge` class extends `BinaryInstruction` and implements basic parallelism: at each instant the two instructions `left` and then `right` are activated in this order. It terminates when both `left` and `right` do terminate.

```
public class Merge extends BinaryInstruction
{
  private byte leftStatus = SUSP, rightStatus = SUSP;

  public Merge (Instruction left, Instruction right)
  {
     super.left = left;
     super.right = right;
  }

  public void reset(){
     super.reset(); leftStatus = rightStatus = SUSP;
  }

  final public String toString(){
     return "(" + left + " || " + right + ")";
  }

  final protected byte activation(Machine machine)
  {
```

```
    if (leftStatus == SUSP) leftStatus = left.activ(machine);
    if (rightStatus == SUSP) rightStatus = right.activ(machine);
    if (leftStatus == TERM && rightStatus == TERM) return TERM;
    if (leftStatus == SUSP || rightStatus == SUSP) return SUSP;
    leftStatus = rightStatus = SUSP;
    return STOP;
  }
}
```

3.4 Atoms

The `Atom` abstract class extends `Instruction` and defines reactive instructions that execute an action and terminate immediately.

```
abstract public class Atom extends Instruction
{
  abstract protected void action(Machine machine);

  final protected byte activation(Machine machine)
  {
      action(machine);
      return TERM;
  }
}
```

`PrintAtom` extends `Atom` to print strings.

```
public final class PrintAtom extends Atom
{
  private String msg;
  public PrintAtom(String msg) { this.msg = msg; }

  final public String toString(){
    return "System.out.print(\"" + msg + "\");";
  }

  final protected void action(Machine machine){
     System.out.print(msg);
  }
}
```

3.5 An Example

The little example we consider consists of running three instants of a machine whose program could be represented by:

```
    (
       stop;{System.out.print("left ")}
    ||
       {System.out.print("right ")}
    );
    {System.out.print("end ")}
```

This code fragment is actually a reactive script.

First, one defines a new machine and an instruction using `Stop`, `Seq`, `Merge`, and `PrintAtom`; then, the instruction is added to the machine program; finally, three machine activations are provoked. The code is:

```
class Example
{
  public static void main (String argv[])
  {
    Machine machine = new Machine();

    Instruction inst =
      new Seq(
        new Merge(
          new Seq(new Stop(),new PrintAtom("left ")),
          new PrintAtom("right ")),
        new PrintAtom("end "));

    machine.add(inst);

    for (int i = 1; i<4; i++){
       System.out.print("instant "+i+": ");
       machine.activ(machine);
       System.out.println("");
    }
  }
}
```

Execution of this class gives:

```
instant 1: right
instant 2: left end
instant 3:
```

Note that termination of the `Merge` only occurs at the second instant because of the `Stop` instruction in the first branch. Note also that printing of `end` occurs in the second instant. Sequencing is "instantaneous", that is control goes to the second component of the sequence as soon as the first one terminates.

3.6 Infinite Loop

SugarCubes provides two kinds of loops: infinite loops and finite ones (described in next section), which both extend `UnaryInstruction`.

When the body of an infinite loop of class `Loop` is terminated, it is automatically and immediately restarted.

A loop is said to be "instantaneous" when its body terminates completely in the same instant it is started. Instantaneous loops are to be rejected because such a loop would never converge to a stable state closing the instant. The `Loop` class detects instantaneous loops at run time, when the end of the loop body is reached twice during the same instant. In this case, the loop stops its execution for the current instant instead of looping for ever during the instant.

```
public class Loop extends UnaryInstruction
{
  protected boolean endReached = false;

  public Loop(Instruction body){ super.body = body; }

  final public String toString(){
    return "loop " + body + " end";
  }

  public void reset(){ super.reset(); endReached = false; }

  final protected byte activation(Machine machine)
  {
    byte res;
    for(;;){
      res = body.activ(machine);
      if (res == TERM){
         if (endReached){
            System.out.println(
               "warning: instantaneous loop detected");
            res = STOP;
            break;
         }
         endReached = true;
         body.reset();
      }else break;
    }
    if (res == STOP){ endReached = false; }
    return res;
  }
```

Note that the loop body is restarted after termination, with the `reset` method.

3.7 Finite Loop

Finite loops of the `Repeat` class execute their body a fixed number of times. Unlike infinite loops, an instantaneously terminating body is not a problem, as it does not prevent the loop to terminate. Therefore, there is no detection of instantaneously terminating bodies of `Repeat` instructions. `Repeat` contains two integer field: `num` which is the initial number of cycles, and `counter` which is the number of cycles performed so far.

```
public class Repeat extends UnaryInstruction
{

  protected int num, counter;

  public Repeat(int num,Instruction body)
  {
    super.body = body;
    this.num = num;
    counter = num;
  }

  public void reset(){ super.reset(); counter = num; }

  final public boolean equals(Instruction inst){
    return  super.equals(inst) && num == ((Repeat)inst).num;
  }

  final public String toString(){
    return "loop {" + num + "} times " + body + " end";
  }

  final protected byte activation(Machine machine)
  {
    while (counter > 0){
     byte res = body.activ(machine);
     if (res == TERM){
       counter-;
       body.reset();
     }else{
       return res;
     }
    }
    return TERM;
  }
```

```
}
```

3.8 Close Instruction

`Close` extends `UnaryInstruction` and executes its body while it is suspended.

```
public class Close extends UnaryInstruction
{
  public Close(Instruction body){ super.body = body; }

  final public String toString(){ return "close " + body; }

  final protected byte activation(Machine machine)
  {
     byte res = SUSP;
     while (res == SUSP){ res = body.activ(machine); }
     return res;
  }
}
```

4 Events

Event based programming is achieved in SugarCubes with the `Event` class of events, and with the class `Config` of event configurations which are boolean expressions of events.

4.1 Events

SugarCubes provide the notion of an event with the following characteristics:

- events are automatically reset at the beginning of each instant; thus, events are not persistent data across instants.
- events can be generated by the `generate` method. This determines the event's presence for the current instant. Generating an event which is already present has no effect.
- an event is perceived in the same way by all parallel components: events are broadcast.
- events can be tested for presence, waited for, or used to preempt a reactive statement.
- one cannot decide that an event is absent during the current instant before the end of this instant (this is the only moment one is sure that the event has not been generated during the instant). Thus, *reaction to absence is always postponed to the next instant.* This is the basic principle of the reactive approach.

To indicate that an event is generated during the current instant, one sets its `generated` field to this instant number. In that way all events are automatically reset at the beginning of each new instant.

The value (present or absent) of an event can only be known safely after the event is fixed, that is as soon as it is generated, or otherwise at the end of the current instant. Event presence values are defined in the `EventConsts` interface.

```
public interface EventConsts
{
  final byte NOT_GENERATED = 0;
  final byte NO_HYPOTHESIS = 0;

  final byte PRESENT  = 1;
  final byte ABSENT   = 2;
  final byte UNKNOWN  = 3;
}
```

The `presence` method returns `PRESENT` if the event is generated, `ABSENT` if it is not (which is known only at the end of the current instant), and `UNKNOWN` otherwise. Class `Event` is:

```
public class Event
implements ReturnCodes, EventConsts, Cloneable
{
  public String name;

  private int generated = NOT_GENERATED;

  public Event(String name){ this.name = name; }

  public boolean isPresent(Machine machine){
    return generated == machine.currentInstant();
  }

  public void generate(Machine machine){
    generated = machine.currentInstant();
  }

  public void makeAbsent(Machine machine){
    generated = - machine.currentInstant();
  }

  public boolean isAbsent(Machine machine){
    return generated == - machine.currentInstant();
  }

  public String toString(){ return name; }
  public Object clone(){ ... }
```

```
  public byte presence(Machine machine)
  {
    if (isPresent(machine)) return PRESENT;
    if (isAbsent(machine) || machine.isEndOfInstant())
       return ABSENT;
    return UNKNOWN;
  }
}
```

4.2 Event Configurations

Event configurations of the class `Config` are boolean expressions of events: a configuration can be the `and`, or the `or` of two events. It can also be the negation `not` of an event, which is true when the event is absent. A configuration is "fixed" when its value can be evaluated safely. For example, a configuration which is a simple event (of the class `PosConfig` defined bellow) is fixed as present as soon as the event is present, or else it is fixed as absent at the end of the current instant.

Abstract class `Config` is:

```
abstract public class Config
implements ReturnCodes
{
  abstract public boolean fixed(Machine machine);
  abstract public boolean evaluate(Machine machine);

  public boolean equals(Config conf){
    return this.getClass() == conf.getClass();
  }
}
```

Unary Configurations

Unary configurations of the abstract class `UnaryConfig` are positive or negative configurations. Positive configurations are just events. Negative configurations are negations of events, corresponding to "`not e`". A unary configuration is fixed when the `presence` method returns a value different from `UNKNOWN`.

```
abstract public class UnaryConfig extends Config
implements EventConsts
{
  protected String eventName;

  public String name(){ return eventName; }
```

```
  public boolean equals(Config config){
     return super.equals(config) &&
            eventName.equals(((UnaryConfig)config).eventName);
  }

  public boolean fixed(Machine machine){
    return (machine.getEvent(eventName)).
                          presence(machine)!=UNKNOWN;
  }
}
```

Positive configurations of class `PosConfig` are just events and evaluation returns true if the event is generated in the machine.

```
public class PosConfig extends UnaryConfig
{
  public PosConfig(String eventName){
     this.eventName = eventName;
  }

  public Event event(Machine machine){
    return machine.getEvent(eventName);
  }

  public String toString(){ return eventName; }

  public boolean evaluate(Machine machine){
     return machine.isGenerated(eventName);
  }
}
```

Negative configurations of class `NegConfig` are negations of events and evaluation returns true if the event is not generated in the machine.

```
public class NegConfig extends UnaryConfig
{
  public NegConfig(String eventName){
     this.eventName = eventName;
  }

  public String toString(){ return "not " + eventName; }

  public boolean evaluate(Machine machine){
    return ! machine.isGenerated(eventName);
  }
}
```

Binary Configurations

Binary configurations are conjunctions (`and`) or disjunctions (`or`) of configurations. The `BinaryConfig` abstract class has two `Config` fields `c1` and `c2`.

```
abstract class BinaryConfig extends Config
{
  protected Config c1,c2;

  public boolean equals(Config config){
    return  super.equals(config) &&
            c1.equals (((BinaryConfig)config).c1) &&
            c2.equals(((BinaryConfig)config).c2);
  }
}
```

A conjunction of the class `AndConfig` is fixed as soon as one component is fixed and evaluates to false: the other one does not need to be also fixed. Otherwise, the conjunction is fixed when both components are. Evaluation returns the "and" of the two components.

```
public class AndConfig extends BinaryConfig
{
  public AndConfig(Config c1, Config c2)
  {
    this.c1 = c1;
    this.c2 = c2;
  }

  public String toString(){
     return "(" + c1 + " and " + c2 + ")";
  }

  public boolean fixed(Machine machine)
  {
    boolean b1 = c1.fixed(machine);
    boolean b2 = c2.fixed(machine);
    if (b1 && !c1.evaluate(machine)) return true;
    if (b2 && !c2.evaluate(machine)) return true;
    return b1 && b2;
  }

  public boolean evaluate(Machine machine){
    return c1.evaluate(machine) && c2.evaluate(machine);
  }
}
```

We do not present here the class `OrConfig` of configuration disjunction as it is very similar to rhe `AndConfig` class.

4.3 Event Declarations

Local events can be declared in an instruction in several ways:

- events declared with `EventDecl` are completely disconnected with the outside, that is external event environment;
- input events declared with `InputDecl` can be generated from outside;
- generations of output event declared with `OutputDecl` are transmitted to outside;
- inputoutput events declared with `InputOutputDecl` can be generated from outside and generations inside their scope are transmitted to outside.

EventDecl Class

`EventDecl` extends `UnaryInstruction` and defines an event which is local to its body and has no connection with the outside. The internal event is stored in the `internal` field.

```
public class EventDecl extends UnaryInstruction
{
  private String internalName;
  private Event internal;

  public EventDecl(String internalName,Instruction body)
  {
    this.internalName = internalName;
    internal = new Event(internalName);
    this.body = body;
  }

  public void reset(){
    super.reset();
    internal = new Event(internalName);
  }

  public Object clone()
  {
    EventDecl copy = (EventDecl)super.clone();
    copy.internal = (Event)internal.clone();
    return copy;
  }

  final public boolean equals(Instruction inst){
```

```
    return  super.equals(inst) &&
      internalName.equals(((EventDecl)inst).internalName);
  }

  final public String toString(){
    return "event " +  internalName + " in " + body + " end";
  }

  final protected byte activation(Machine machine)
  {
    Event save = machine.getEvent(internalName);
    machine.putEvent(internalName,internal);
    byte res = body.activ(machine);
    machine.putEvent(internalName,save);
    return res;
  }
}
```

IODecl Class

Abstract class `IODecl` extends `UnaryInstruction` and defines a scope in which a local event with name `internalName` is linked to an external event with name `externalName`. The `setInternal` method sets the `internalName` event in the local context, and `saveInternal` restores its old value.

```
abstract public class IODecl extends UnaryInstruction
{
  protected String internalName;
  protected String externalName;
  protected Event internal = new Event(internalName);
  protected Event save;

  public void reset(){
     super.reset(); internal = new Event(internalName);
  }

  public Object clone(){ ... }

  final public boolean equals(Instruction inst){
    return  super.equals(inst) &&
      internalName.equals(((IODecl)inst).internalName) &&
      externalName.equals(((IODecl)inst).externalName);
  }
```

```
  protected void setInternal(Machine machine){
    save = machine.getEvent(internalName);
    machine.putEvent(internalName,internal);
  }

  protected void saveInternal(Machine machine){
    internal = machine.getEvent(internalName);
    machine.putEvent(internalName,save);
  }
}
```

InputDecl Class

Class `InputDecl` extends `IODecl` and define an input event.

```
public class InputDecl extends IODecl
{
  public InputDecl(String internalName,String externalName,
                                        Instruction body)
  {
    this.internalName = internalName;
    this.externalName = externalName;
    this.body = body;
  }

  final public String toString(){
    return "input "+internalName+
           " is "+externalName+" in "+body+" end";
  }

  final protected byte activation(Machine machine)
  {
    boolean present = machine.isGenerated(externalName);
    setInternal(machine);           // Set internal event
    if (present) machine.generate(internalName); // Transmit
    byte res = body.activ(machine); // Execute body
    saveInternal(machine);          // Save internal event
    return res;
  }
}
```

`OutputDecl` and `InOutDecl` classes are very similar, and for simplicity we do not give their definitions here.

5 Event Based Instructions

In this section we introduce instructions related to events: generation of an event, control of an instruction by the presence of an event, waiting for an event configuration, preemption of an instruction, and presence test of an event.

5.1 Event Generation

The `Generate` class extends `Atom` which means that event generation terminates instantaneously. Generating an event in a machine calls the machine `newMove` method to indicate that something new happens in the system; thus, instructions waiting for the event (see next 5.3) will have the possibility to see it as present during the current instant.

```
public class Generate extends Atom
{
  private String eventName;

  public Generate(String eventName){
    this.eventName = eventName;
  }

  final public boolean equals(Instruction inst){
    return  super.equals(inst) &&
      eventName.equals(((Generate)inst).eventName);
  }

  final public String toString(){
    return "generate " + eventName;
  }

  final protected void action(Machine machine){
    Event event = machine.getEvent(eventName);
    machine.newMove();
    event.generate(machine);
  }
}
```

5.2 Control Instruction

`Control` extends `UnaryInstruction` and controls its body by the presence of an event: the body is run only during the instants where the event is present.

```
public class Control extends UnaryInstruction
implements EventConsts
{
```

```
  private String eventName;

  public Control(String eventName,Instruction body){
    this.eventName = eventName;
    this.body = body;
  }

  final public boolean equals(Instruction inst){
    return  super.equals(inst)
            && eventName.equals(((Control)inst).eventName);
  }

  final public String toString(){
    return "control " + body + " by " + eventName;
  }

  final protected byte activation(Machine machine)
  {
    Event event = machine.getEvent(eventName);
    switch(event.presence(machine)){
    case PRESENT: return body.activ(machine);
    case ABSENT:  return STOP;
    default: return SUSP;
    }
  }
}
```

5.3 Waiting for Events

`Await` extends `Instruction` and contains a `Config` field. The `activation` method returns `SUSP` while the configuration is not fixed, then it evaluates it. If evaluation returns false, meaning that the configuration waited for is not satisfied, then the method stops. If evaluation returns true, meaning that the configuration waited for is satisfied, then the `Await` terminates and returns `TERM` if the end of the current instant is not already reached, `STOP` otherwise. For example evaluation of the configuration corresponding to "`not e`" returns true if `e` was not generated, and `activation` returns `STOP` in this case. This is coherent with the basic principle of 4.1 which states that the absence of an event cannot be decided before the end of the current instant.

```
public class Await extends Instruction
{
  private Config config;

  public Await(Config config){ this.config = config; }
  final public String toString(){ return "await " + config; }
```

```
  final public boolean equals(Instruction inst){
    return  super.equals(inst) &&
            config.equals(((Await)inst).config);
  }

  final protected byte activation(Machine machine)
  {
    if (!config.fixed(machine)) return SUSP;
    if (!config.evaluate(machine)) return STOP;
    terminate();
    return machine.isEndOfInstant() ? STOP : TERM;
  }
}
```

5.4 Preemption

`Until` extends `BinaryInstruction` and implements preemption: execution of the `left` instruction, called the "body", is aborted when an event configuration becomes true. One says then that `left` is "preempted" by the configuration and, in this case, control goes to `right` which is called the "handler".

The preemption implemented by `Until` is "weak": `left` is not prevented to react at the very instant of preemption.

```
public class Until extends BinaryInstruction
{
  private Config config;
  private boolean activeHandle = false;
  private boolean resumeBody = true;

  public Until(Config config,Instruction body,
                             Instruction handler){
    this.config = config; left = body; right = handler;
  }

  public Until(Config config,Instruction body){
    this.config = config; left = body; right = new Nothing();
  }

  public void reset()
  {
    super.reset();
    activeHandle = false;
    resumeBody = true;
  }
```

```
  final public boolean equals(Instruction inst){
    return  super.equals(inst)
           && config.equals(((Until)inst).config);
  }

  final public String toString(){
    if (right instanceof Nothing)
        return "do "+left+" until "+config;
    return "do "+left+" until "+config+" actual "+right+" end";
  }

  final protected byte activation(Machine machine)
  {
    if (activeHandle) return right.activ(machine);
    if (resumeBody){ // body is to be executed
      byte res = left.activ(machine); // weak preemption !
      if (res != STOP) return res;
      resumeBody = false;
    }
    if (!config.fixed(machine)) return SUSP;
    if (config.evaluate(machine)){ // actual preemption
      activeHandle = true;
      if (machine.isEndOfInstant()) return STOP;
      return right.activ(machine);
    }
    resumeBody = true; // to re-execute the body at next instant
    return STOP;
  }
}
```

5.5 Event Presence Test

`When` extends `BinaryInstruction` and gives the control to the left or right instruction according to an event configuration. Evaluation of the configuration only takes place at the instant the control reaches the `When` instruction and the choice of the branch to be executed is final.

```
public class When extends BinaryInstruction
{
  private Config config;
  private boolean confEvaluated = false;
  private boolean value;

  public When(Config config,Instruction th,Instruction el)
```

```
  {
    this.config = config;
    left = th;
    right = el;
  }

  public void reset(){ super.reset(); confEvaluated = false; }

  final public boolean equals(Instruction inst){
    return  super.equals(inst)
          && config.equals(((When)inst).config);
  }

  final public String toString(){
    return "when "+config+" then "+left+" else "+right+" end";
  }

  final protected byte activation(Machine machine)
  {
      if (!confEvaluated){
          if (!config.fixed(machine)) return SUSP;
          confEvaluated = true;
          value = config.evaluate(machine);
          if(machine.isEndOfInstant()) return STOP;
      }
      return value ? left.activ(machine) : right.activ(machine);
  }
}
```

5.6 An Example

In this example, one first adds an instruction which waits for an event named `e` and then prints "`e!`" to a machine . The machine is run and a copy of the previous instruction is also added to it. Then, the machine is run for the second time. Finally, an instruction which generates `e` is added, and the machine is run for the third time.

```
class Example1
{
  static int i = 1;

  static void run(Machine machine){
    System.out.print("instant "+(i++)+": ");
    machine.activ(machine);
    System.out.println("");
  }
```

```
  public static void main (String argv[])
  {
    Machine machine = new Machine();

    Instruction inst =
      new Seq(
        new Await(new PosConfig("e")),
        new PrintAtom("e! "));

    machine.add(inst);
    run(machine);
    machine.add((Instruction)inst.clone());
    run(machine);
    machine.add(new Generate("e"));
    run(machine);
  }
}
```

Execution gives:

```
instant 1:
instant 2:
instant 3: e! e!
```

Note that the two `Await` instructions are both fired during the same third instant where the event is generated (events are broadcast). Note also the use of the `clone` method to get a copy of the instruction.

6 Conclusion

We have presented an overview of SugarCubes to program reactive instructions (`Instruction` class) run in parallel by reactive machines (`Machine` class) and communicating with broadcast events (`Event` class).

Several implementations using SugarCubes are presented in three companion papers which are available at `http://www.inria.fr/meije/rc/SugarCubes`:

- The SugarCubes Tool Box - Nets of Reactive Processes Implementation
- The SugarCubes Tool Box - rsi-java Implementation
- The SugarCubes Tool Box - Icobj Programming Implementation

Index

Unité de recherche INRIA Lorraine, Technopôle de
Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101,
54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de
Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de
l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau,
Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia Antipolis, 2004 route des
Lucioles, BP 93, 06902 SOPHIA ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt,
BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399